PR72 - Logistics Platform Project

Computer Science - UTBM

Academic Year: 2024 - 2025

Project Objective

Develop a `C++` simulator for a logistics platform, from receiving goods to delivering them to the customer, while managing product flows and storage of goods.

Student: Guillaume Schneider

University of Technology of Belfort-Montbéliard (UTBM)

In the context of developing a logistics platform simulator, I have chosen to implement a real-time simulation. This type of simulation presents specific challenges, requiring the integration of advanced techniques adapted to the real-time nature of the simulated processes.

The first dimension of this project is based on the use of **multithreading** to orchestrate the simultaneity of different actions within the simulation. For example, time cycle management will allow synchronization of operations, while simulations of tasks performed by employees will realistically simulate their respective actions.

In parallel, a fine simulation is envisaged, taking into account:

- A **spatial dimension**, with the management of addresses and coordinates to model the flow of goods and distances between different logistical points;
- A **temporal dimension**, ensuring that events unfold according to a realistic chronological framework;
- A **realistic data model**, integrating parameters such as storage capacities, product arrival frequencies, and transport resource constraints.

Finally, particular attention is paid to **best practices** and the **ease of use** of the project. This includes:

- The use of **CMake** for compilation management;
- The ability to configure the simulator via a **CLI** (Command Line Interface);
- The implementation of an **initialization** mechanism to facilitate the quick start of the simulation.

Contents

1 Multithreading **1**

1.1 Multithread Model 1

1.1.1 Thread Logic and Synchronization 2

1.2 Unitary Action System 3

1.2.1 Employee Activity Distribution System 3

1.2.2 Use of the Strategy Pattern and Dependency Injection 4

1.3 Scheduler 6

1.3.1 General Operation 6

1.3.2 Role and Responsibilities 6

1.3.3 Synchronization Management 6

1.4 Real-Time Shell Functionalities 6

1.5 Error and Command Management 7

1.6 Multithread Operation of the Shell 7

1.7 Integration into the Multithread System 8

2 Life-Size System **9**

2.1 Spatial and Temporal Dimensions 9

2.1.1 Spatial Dimension 9

2.1.2 Temporal Dimension 9

2.2 Data Model 10

2.2.1 Abstract and Concrete Data 10

2.2.2 Data Management with Singleton and JSON 11

2.2.3 Unique Key for Relationship Management 11

2.2.4 Product Reference Based on Concatenation 11

2.2.5 Visual Representation of the Data Model 12

2.3 Parameters and CLI 14

3 Logistics Platform Logic **15**

3.1 Representation Model 15

3.2 Execution Sequence 15

3.3 Global Workflow 19

4 Feedback and Conclusion **21**

Appendix: Class Diagram **22**

1 Multithreading

One of the major aspects of this simulation is the integration of multithreading, an essential approach to ensuring realistic and efficient operation of the real-time logistics platform. This technique allows for the simultaneous execution of several tasks related to the management of employees, tools, and resources, while ensuring effective synchronization of the various processes.

1.1 Multithread Model

To best represent the activities of employees, each employee is associated with a dedicated thread. This model allows for the simultaneous execution of tasks by multiple employees, thus providing a realistic and fluid simulation of daily operations. Additionally, a specific thread is dedicated to displaying the activities of employees, ensuring continuous and synchronized updates of the information visible to the user.

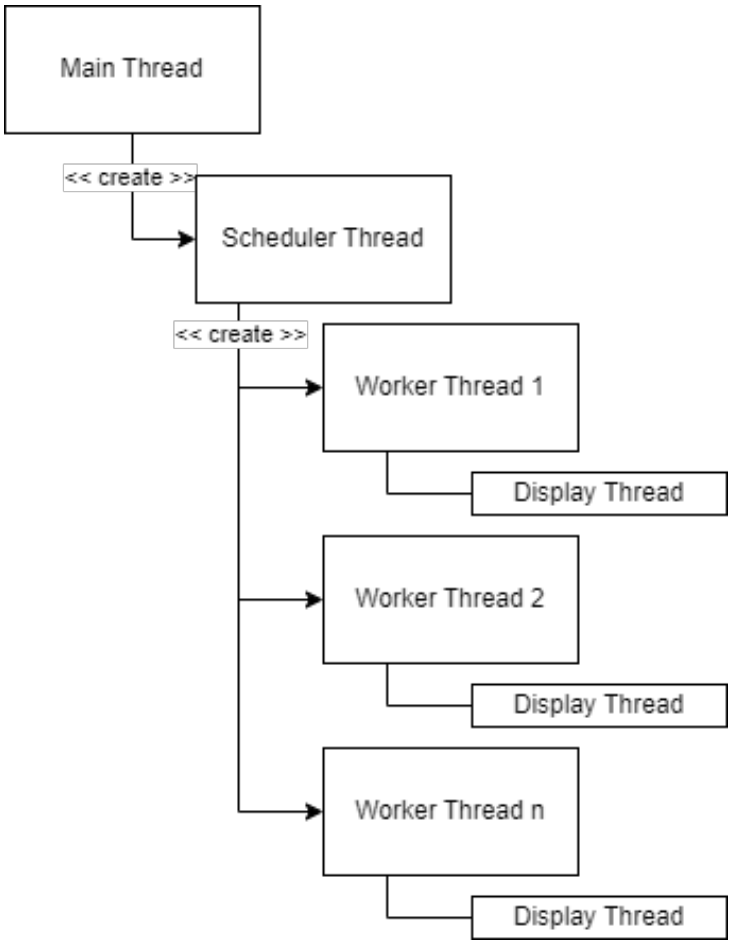

Figure 1: Thread hierarchy in the simulation.

1.1.1 Thread Logic and Synchronization

The logic of the threads, presented in the following diagram (Fig. 2), shows how tasks are distributed by the **Scheduler** to available **Workers**. Synchronization of the threads is ensured using mutexes to avoid access conflicts, particularly when updating the state or display.

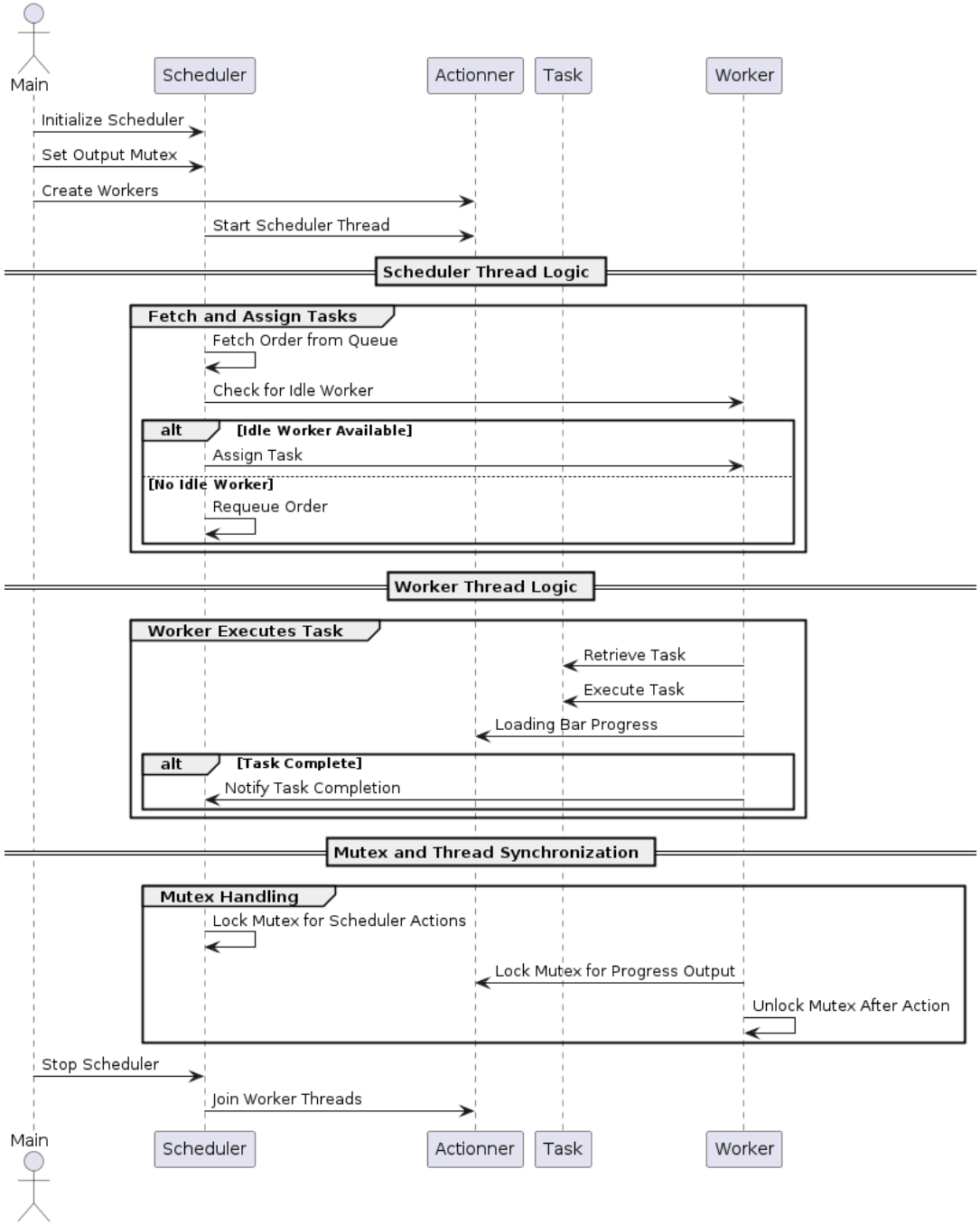

Figure 2: Thread logic and mutex synchronization.

Detailed Explanations:

- **Scheduler Thread Logic**: The scheduler retrieves a task from the queue and searches for an available employee. If an employee is available, the task is assigned to them. Otherwise, the task is reinserted into the queue.
- **Worker Thread Logic**: Once a task is assigned, the employee executes it and informs the scheduler upon completion. A progress bar is displayed to visualize the progress.
- **Mutex and Synchronization**: Mutexes are used to manage critical actions, such as task distribution by the scheduler and the display of progress bars. This ensures secure management of shared resources.

1.2 Unitary Action System

In the simulation, a unitary action system has been designed to manage the distribution and execution of employee activities. This system relies on several key concepts to ensure efficient and modular organization.

1.2.1 Employee Activity Distribution System

The central philosophy of the employee activity system is based on a modular and flexible organization, capable of meeting the specific requirements of a simulated logistics platform. This system is structured around fundamental principles that allow for the efficient management of task distribution and execution while ensuring a clear separation of responsibilities.

First, the **temporal dimension** plays a key role in this system. Although not yet fully implemented, this dimension aims to integrate notions of deadlines, priorities, and pacing in activity management. The objective is to ensure that critical tasks are executed as a priority while respecting realistic temporal constraints. This will optimize logistical flows by reducing employee and resource downtime.

Next, a **strict separation between activity logic and the employees themselves** is implemented. Employees act as independent executors, taking on the tasks assigned to them without interfering with the allocation mechanisms. This architecture decouples the "what to do" from the "how to do it," thus facilitating the extension and maintenance of the system.

Moreover, the system is designed to allow for a **great diversity of activities**. Each employee can take on several types of tasks, whether it be handling, transport, or product organization. This versatility is essential to reflect the reality of modern logistics platforms, where employees often have to juggle different responsibilities.

1.2.2 Use of the Strategy Pattern and Dependency Injection

The system relies on the **Strategy** pattern, which allows for the definition of interchangeable action blocks for each employee. This mechanism is based on a **state machine** logic, where each state corresponds to a specific action that the employee must execute in a defined cycle.

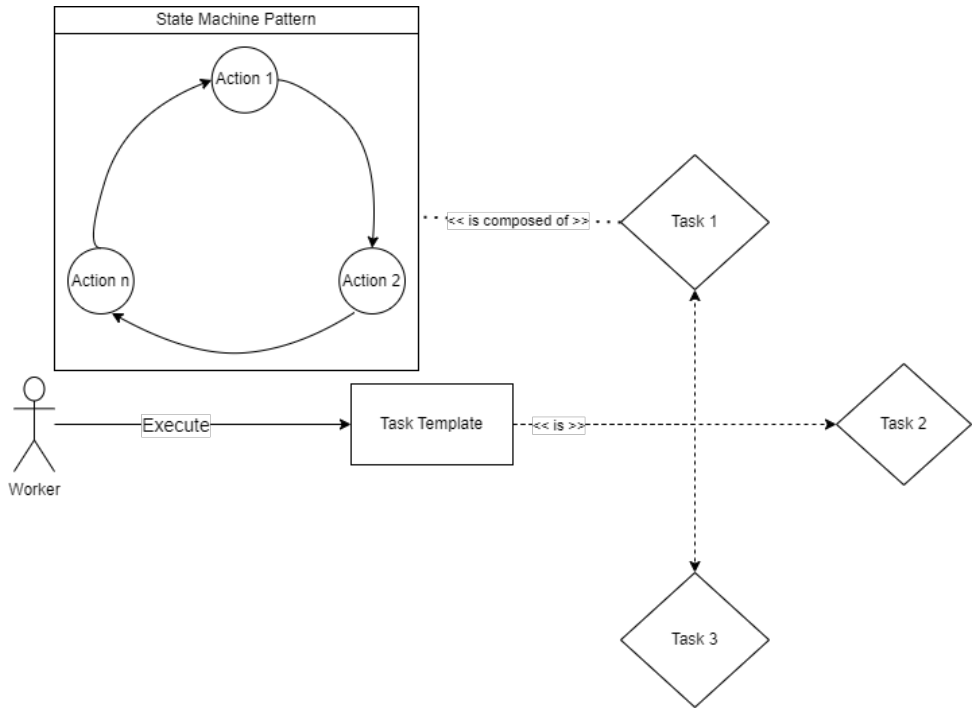

Figure 3: State machine pattern and strategy for task management.

As shown in the diagram of Fig. 3:

- The **state machine** manages transitions between actions (e.g., Action 1 to Action 2, or Action 2 to Action n).
- Each task is modeled as an instance of `Task Template`, allowing for the representation of customized activities through a generic structure.
- Actions are executed following the defined model, ensuring an ordered execution of tasks and flexibility for adding new activities.

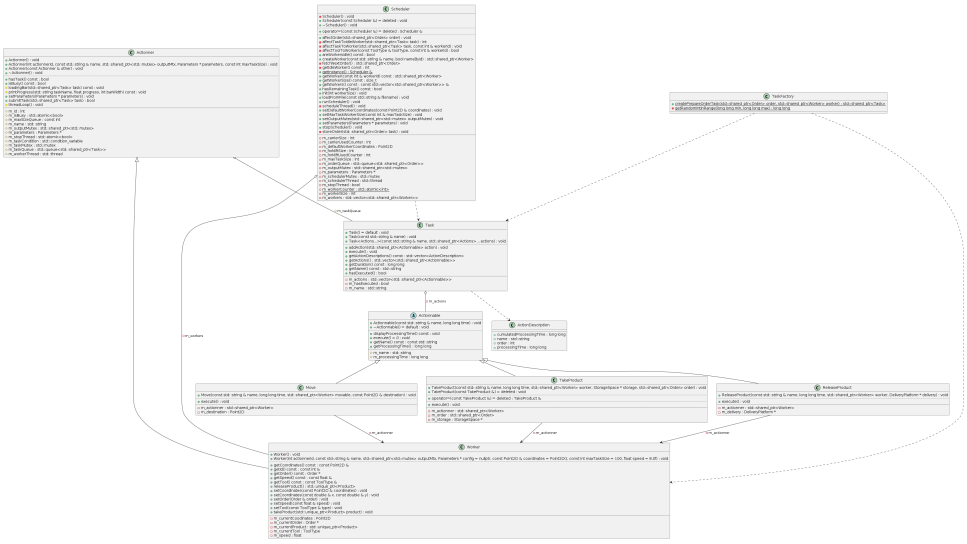

Figure 4: Class diagram of the pattern (zoom in for details).

1.3 Scheduler

The scheduler is a central element in managing tasks and resources in the simulation. Its design is based on the idea of providing precise and flexible control over all operations performed by workers and tools within the logistics platform.

1.3.1 General Operation

The scheduler operates in its own thread throughout the application's execution. This choice allows for a dedicated process to manage tasks, ensuring smooth execution without interference from other simulation components.

1.3.2 Role and Responsibilities

The scheduler is responsible for:

- **Assigning Orders**: The scheduler identifies pending orders and assigns each task to an available worker (represented by a worker thread). This dynamic management ensures optimal resource allocation.
- **Creating and Managing Resources**: The scheduler is also responsible for creating and configuring workers, machines, and tools needed to execute tasks. This includes initializing associated worker threads and setting up communication mechanisms between them.

1.3.3 Synchronization Management

To ensure coherent execution and avoid access conflicts, the scheduler uses mutexes. These allow for the synchronization of:

- Passing orders and parameters to workers.
- Interactions with tools or machines shared by multiple workers.

1.4 Real-Time Shell Functionalities

The Shell is a dynamic interface that allows interaction with the simulation in real time. With this tool, it is possible to execute several actions directly during the simulation's execution. For example, users can query the database to obtain information about stored products, ongoing orders, or the system's overall status via commands such as `list products`, `list orders`, or `status`.

One of the main strengths of the Shell is its ability to introduce flexibility compared to compiled solutions. Indeed, it is possible to add instructions dynamically, without

restarting or recompiling the program. For example, a command can be used to disable progress bars (`disable loading`) or to modify certain parameters in real time. This functionality makes the Shell particularly suitable for evolving or experimental simulation environments.

Finally, the Shell offers interactive configuration options. Users can customize the display or directly control the various simulation modules. These interactions are done via simple and accessible commands, making the use of the simulation fluid and intuitive.

1.5 Error and Command Management

The robustness of the Shell is based on a well-thought-out error management mechanism. Before execution, all commands entered by the user are analyzed to verify their validity and syntax. For example, if a command like `list product` is entered instead of `list products`, the Shell returns an explicit error message to guide the user.

Additionally, some commands have specific validations. A command like `list product [id]` requires a valid product identifier, while `list orders` does not tolerate additional arguments. This control ensures not only the security of interactions with the simulator but also a better user experience.

The Shell also integrates a command history. Users can navigate through previous commands using the keyboard arrows, simplifying the reuse of frequently used commands. These features enhance the Shell's interactivity while limiting operational errors.

1.6 Multithread Operation of the Shell

The Shell operates independently thanks to its dedicated thread, which manages user input capture and real-time output display. This design ensures that the Shell remains responsive, even when other system threads, such as workers or the scheduler, are heavily loaded. User inputs, although blocking, do not interfere with the overall execution thanks to rigorous management of shared resources.

When a command is executed, the Shell can interact directly with other threads, such as the scheduler or workers. For example, a `status` command triggers communication with the scheduler to collect and display the task status. This communication model relies on fine synchronization of resources using shared mutexes, ensuring the order and security of exchanges.

Finally, the Shell can modify atomic variables to influence the simulator's global state. A command like `exit` updates a shared variable that signals all threads to terminate their tasks and shut down properly. This approach ensures smooth and secure management of the simulator's shutdown.

1.7 Integration into the Multithread System

The Shell's architecture integrates perfectly into the simulation's multithread model. By sharing the same synchronization mechanisms as the scheduler or workers, the Shell ensures coherent execution in environments where resources are heavily loaded. For example, dynamic displays of progress bars or status returns use centralized mutexes to avoid write conflicts.

The Shell also works in harmony with the scheduler to distribute tasks or interact with the command queue. When a user executes a command that requires modifying the system, such as displaying remaining tasks, the Shell communicates with the scheduler while respecting synchronization constraints.

2 Life-Size System

The life-size system is designed to reflect a realistic modeling of temporal and spatial dimensions within the simulation framework. This approach allows for the faithful representation of physical and temporal constraints associated with logistical flows.

2.1 Spatial and Temporal Dimensions

The spatial and temporal dimensions constitute the two fundamental axes of the simulation, allowing for the realistic representation of geographical and temporal relationships within the logistics platform.

2.1.1 Spatial Dimension

The spatial dimension is at the heart of representing geographical relationships and the movements necessary for managing logistical flows. It is based on the following principles:

- **2D Coordinate System**: The addresses of different logistical points (sites, customers, warehouses) are associated with two-dimensional coordinates, expressed in kilometers. This allows for the modeling of spatial relationships and the calculation of distances between points.
- **Worker Speeds**: Each worker has a specific speed, used to evaluate their travel times between different locations.
- **Travel Time Calculations**: Travel durations are determined based on 2D coordinates and worker speeds, faithfully reflecting the physical constraints associated with movements.
- **Address Hashing**: Addresses are associated with unique identifiers obtained through a hashing system. This optimizes searches and allows for efficient management of coordinates and spatial relationships.

2.1.2 Temporal Dimension

The temporal dimension focuses on managing durations and deadlines for simulation activities, integrating dynamic and chronological constraints. It is based on the following elements:

- **Processing Times**: Each action (movement, handling, storage) is associated with a specific duration, measured in milliseconds, and defined based on realistic data.
- **Realistic Data-Based Calculations**: Execution times are calculated based on worker speeds and distances to be traveled. For example, for a movement, the duration is proportional to the distance traveled divided by the worker's speed.

- **Synchronization and Visualization**: A command-line progress bar is used to visualize the progress of actions in real time, ensuring dynamic and interactive tracking of tasks.
- **Adjustable Time Scale (*timescale*)**: The system allows for the definition of a time scale factor, where one real second can correspond to several simulated seconds. For example, a configuration `timescale = 5` means that 1 real second equals 5 simulated seconds. This flexibility is particularly useful for speeding up or slowing down simulations based on analysis needs.
- **Stop Conditions**: Two main mechanisms govern the stopping of operations:
  1. The scheduler can decide to interrupt a task or a set of operations.
  2. A predefined time limit for a task can be reached, leading to its automatic termination.
- **Non-Deterministic Algorithm**: To reflect real-world uncertainties, the system includes a non-deterministic algorithm. For example, product retrieval times or other delays can vary randomly, simulating logistical uncertainties.

2.2 Data Model

The data model constitutes the foundation on which the simulation is built, allowing for the efficient structuring and management of abstract and concrete data necessary for the proper functioning of the system. This model is designed to be flexible, extensible, and consistent with logistical requirements.

2.2.1 Abstract and Concrete Data

The data model distinguishes two main types of data: abstract data and concrete data, each with specific properties and management adapted to their role in the simulation.

Abstract Data Abstract data represents general and constant concepts, such as product references or addresses. This data is often copied, as it does not undergo any modification during the program's execution. Its immutability exempts it from the use of references or pointers, thus simplifying memory management. This choice ensures increased efficiency while avoiding risks associated with unexpected data modifications.

Concrete Data Conversely, concrete data corresponds to specific instances that must be modified during the simulation. This data, such as orders, products, or coordinates, must remain in memory for the entire duration of the program to ensure accessibility. To ensure efficient management of this data, unique or shared references are used:

- `std::unique_ptr`: Used, for example, for products, each instance is associated with a unique reference. This ensures that a product can only be manipulated by a single holder, avoiding conflicts.
- `std::shared_ptr`: Used in other cases where multiple components need simultaneous access to the same concrete data.

2.2.2 Data Management with Singleton and JSON

To ensure coherent and efficient data management within the simulation, the data model relies on two main approaches: the use of the Singleton pattern for databases and the adoption of the JSON format for data storage and exchange. Databases are designed as singletons, ensuring that only one instance of each database exists throughout the simulation's execution. This approach offers several advantages, including a unique lifetime for data, global accessibility for all system components, and simplified maintenance through the centralization of shared information. This avoids memory duplication and facilitates interactions between different modules, such as workers, the scheduler, or tools.

Additionally, data is stored and exchanged in JSON format. This format allows for smooth interaction between humans and machines, enabling users to modify configuration files or analyze simulation results without major technical difficulties. Moreover, JSON provides a persistent storage solution, allowing for the saving of intermediate or final simulation states while ensuring data traceability. Thanks to its horizontal structure, it efficiently represents relationships between entities (such as orders, addresses, or products) without introducing unnecessary complexity. Together, these two approaches ensure centralized, efficient, and tailored data management to the specific needs of a complex logistical simulation.

2.2.3 Unique Key for Relationship Management

Entities in the data model are identified by unique keys, represented as strings (`std::string`). This approach offers several advantages:

- Ensures unique identification of entities in different databases.
- Allows for the efficient retrieval of associated data using identifiers, similar to primary keys in a relational database.

2.2.4 Product Reference Based on Concatenation

Product references are constructed from the concatenation of the following attributes:

$$\text{Product Reference} = \texttt{category} + \texttt{subCategory} + \texttt{year} + \texttt{reference}$$

Example: If a product belongs to the category `"EL"`, sub-category `"PH"`, was created in `"24"`, and has the unique identifier `"00123"`, the product reference will be generated as follows:

$$\text{Product Reference} = \texttt{"ELPH2400123"}$$

The product's serial number is then generated by combining this reference with a counter indicating the number of products created in the same category and sub-category. For example:

$$\text{Serial Number} = \texttt{Product Reference} + \texttt{counter}$$

With a counter equal to `5`, the serial number becomes:

$$\text{"ELPH24001235"}$$

2.2.5 Visual Representation of the Data Model

The visual representation of the data model reflects the entire philosophy of its design, which combines two complementary approaches. On the one hand, it relies on the use of primary keys and unique identifiers to structure relationships between entities, ensuring coherent and traceable data management. On the other hand, the model adopts a horizontal format, like JSON, to simplify storage and offer increased flexibility.

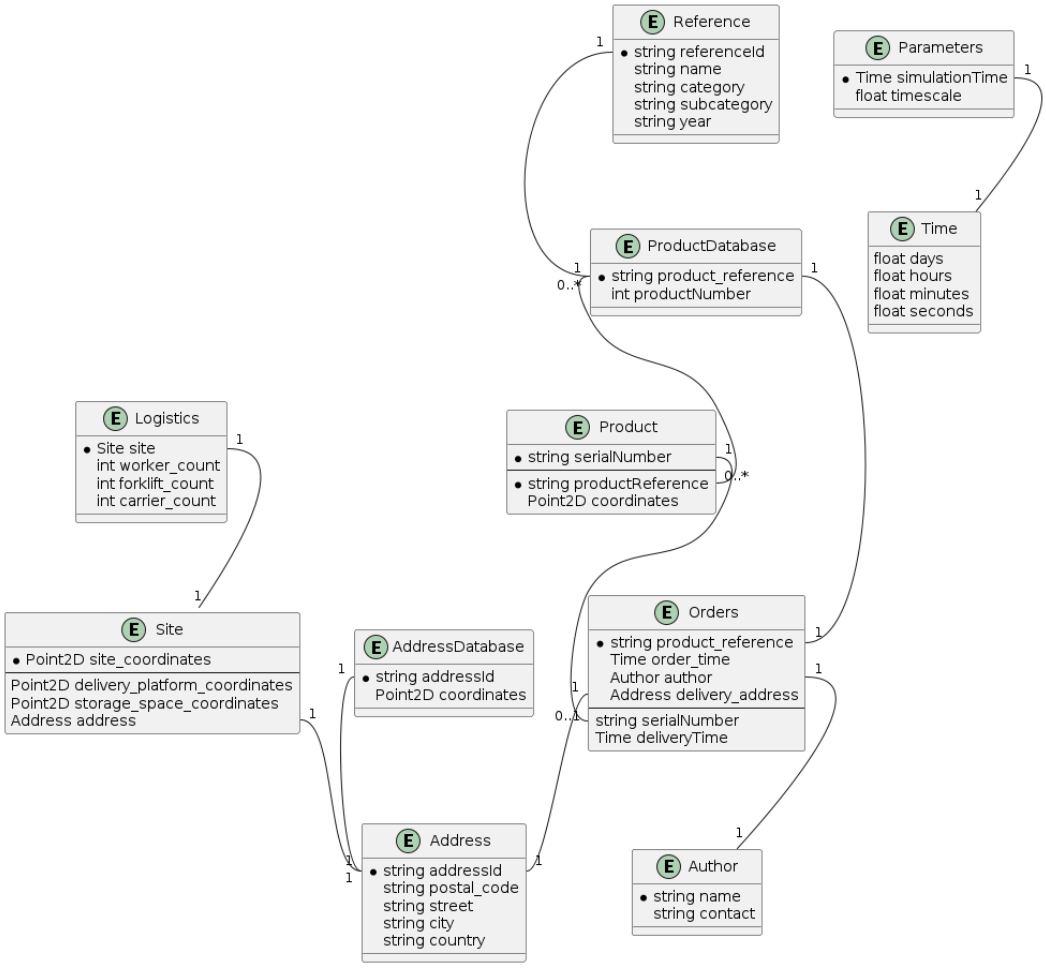

Figure 5: Data model and relationships between entities.

2.3 Parameters and CLI

The management of parameters and the use of a command-line interface (CLI) play a key role in the initialization and configuration of the simulation.

On the one hand, the initialization system is designed to automatically configure singletons and databases. This includes initializing basic parameters, ensuring that all entities necessary for the simulation's execution are correctly defined and accessible.

On the other hand, a parameterized command-line interface allows for the specification of a configuration file when launching the simulation. This file, indicated using the `-config` argument followed by the file name, contains the information necessary to link databases and define the system's global parameters. For example, it provides paths to database files, ensuring flexible and customizable configuration for each execution. A typical command might be:

```
./simulator -config config.json
```

3 Logistics Platform Logic

3.1 Representation Model

The representation model of the logistics platform is based on a modular structure that distinguishes three main entities: workers, the logistics site, and customers. This approach allows for the realistic modeling of interactions and flows within the simulation.

Worker

Workers play a central role in managing logistical operations. They are equipped with specific tools, such as conveyors or forklifts, which allow them to perform various tasks, such as moving and handling products. These tools ensure better productivity and reflect the real operational needs of logistics platforms.

Site

The logistics site is structured into two main areas:

- **Storage Area**: This area is dedicated to storing products awaiting processing or delivery. It is organized to optimize space usage and facilitate access to merchandise.
- **Delivery Platform**: This area is dedicated to preparing and shipping orders. It includes delivery trucks, each capable of containing multiple products destined for customers. These trucks allow for the simulation of loading and transporting merchandise.

Customer

Customers are represented by their coordinates and addresses, which serve to identify the delivery points for orders. This abstraction allows for the integration of distribution flows while ensuring efficient management of relationships between the logistics site and final recipients.

3.2 Execution Sequence

The operation of the logistics platform is based on managing movements and tasks, using a state machine logic. Each sequence models the paths taken by workers based on the actions they perform. These sequences allow for the representation of logistical processes while clearly differentiating physical movements from executed actions.

Sequence 1: Product Management on the Site

This sequence describes the steps necessary for managing products within the logistics site:

- Move to the storage area.
- Pick up the product to be stored.
- Move to the delivery platform.
- Drop off the product on the delivery platform.
- Return to the site base.

This sequence is activated if the following conditions are met:

- An employee is available.
- A product management task is pending.

Sequence 2: Order Delivery

The second sequence illustrates the operations necessary for delivering products to customers:

- Move to the customer.
- Deliver the order (repeated for each order contained in the truck).
- Return to the logistics site.

This sequence is activated if the following conditions are met:

- A truck and an employee are available.
- The truck is filled with products to be delivered.

Remarks on Sequence Diagrams

The presented diagrams do not describe the specific actions of employees but rather the physical paths taken during these actions. They allow for the visualization of movements between different entities (e.g., storage area, delivery platform, customer). To obtain a representation of the actions themselves, it would be necessary to replace the entities with arrows representing transitions or actions.

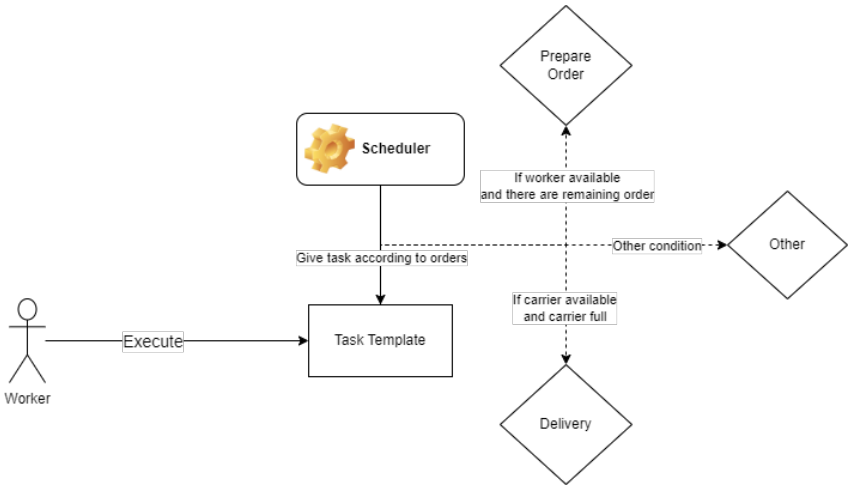

Figure 6: Diagram of the worker task and action system.

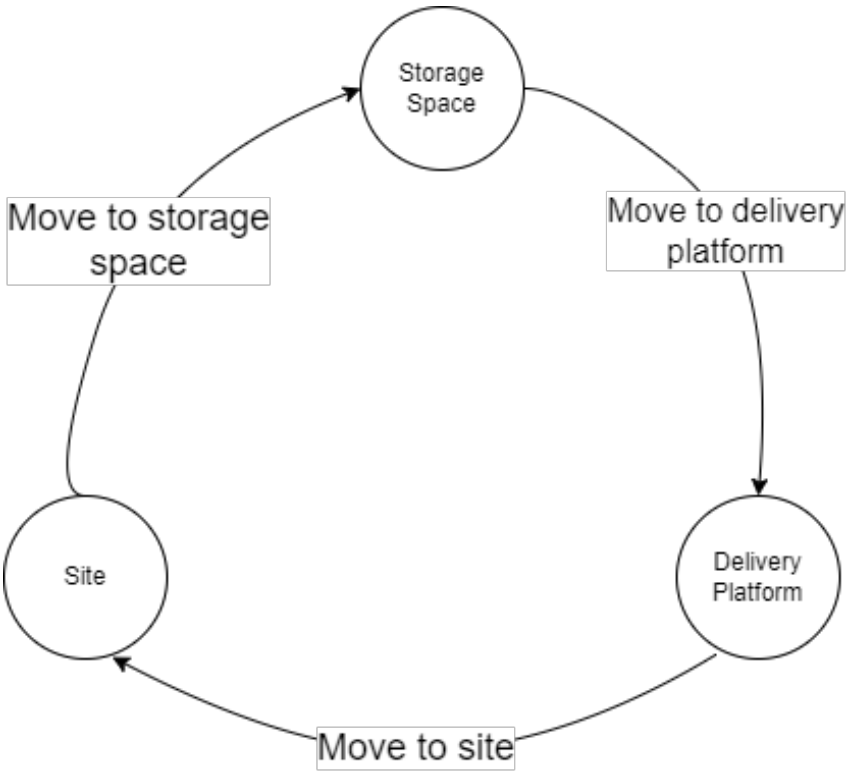

Figure 7: Diagram of the order preparation sequence.

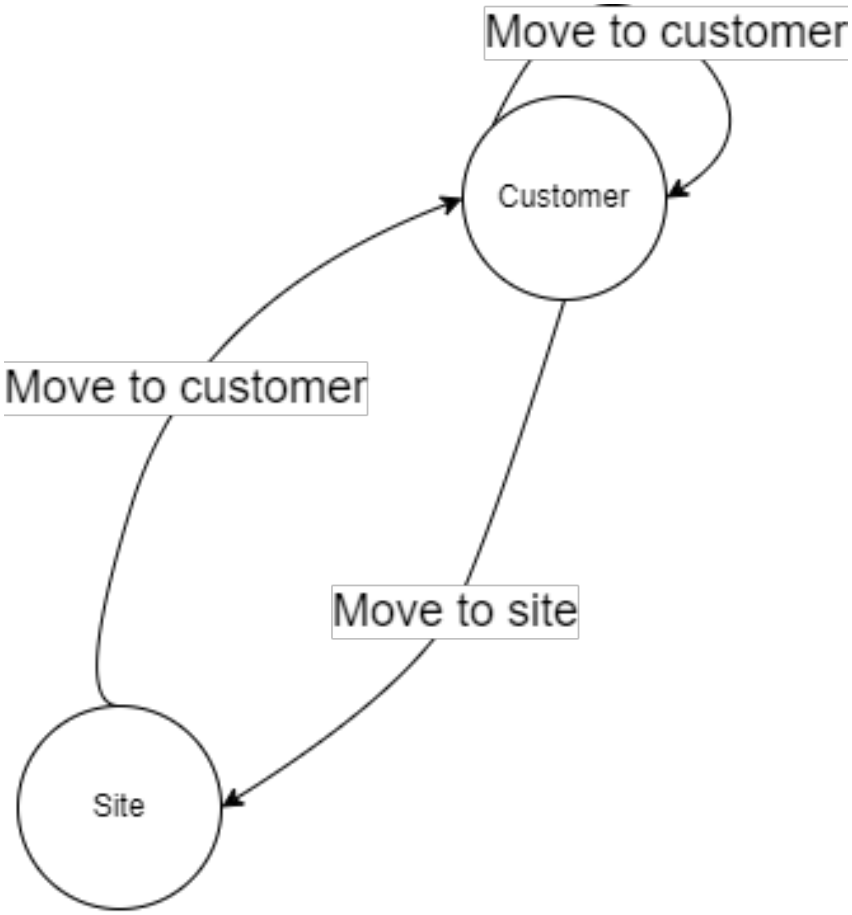

Figure 8: Diagram of the order delivery sequence.

3.3 Global Workflow

The global workflow of the simulation follows a series of logical steps, from initialization to termination, to ensure smooth and efficient operation. These steps cover preparation, execution, and finalization of the simulated processes.

a. Initialization and Verifications

Initialization is the crucial first step in setting up the simulation environment. It includes:

- **Clearing the Console**: Resetting the user interface via a `clear()` command.
- **Creating a Shared Mutex**: Synchronizing outputs to avoid simultaneous access conflicts.
- **Injecting Arguments**: Using the `Initializer` to integrate configuration parameters passed via the command line.
- **Loading Necessary Data**: Importing references, orders, parameters, scheduling, and site-related information.

b. Database Configuration

This step aims to configure the databases used by the simulation:

- **Managing References**: Setting up product references via the `ReferenceManager`.
- **Managing Orders**: Initializing and structuring orders with the `OrderDatabase`.
- **Initializing Addresses**: Associating geographical coordinates with addresses in the `AddressDatabase`.

c. Planning and Resource Allocation

Planning is a key step in organizing and allocating resources optimally:

- **Converting Durations to Seconds**: Standardizing time scales for coherent management.
- **Verifying the Time Scale**: Validating the configuration with the `verifyTimescale` function.
- **Distributing Orders**: Assigning tasks to workers via the scheduler (`Scheduler`).

d. Simulation Execution

This step corresponds to the actual execution of the simulation:

- **Calculating Simulated Duration**: Determining the total simulation time based on defined parameters.
- **Simulation Loop**: Managing real time and controlling remaining tasks until completion.
- **Stopping the Scheduler**: Putting it on hold when all tasks are completed or workers are inactive.

e. Termination

The simulation ends with the stopping of the scheduler to finalize operations and release resources.

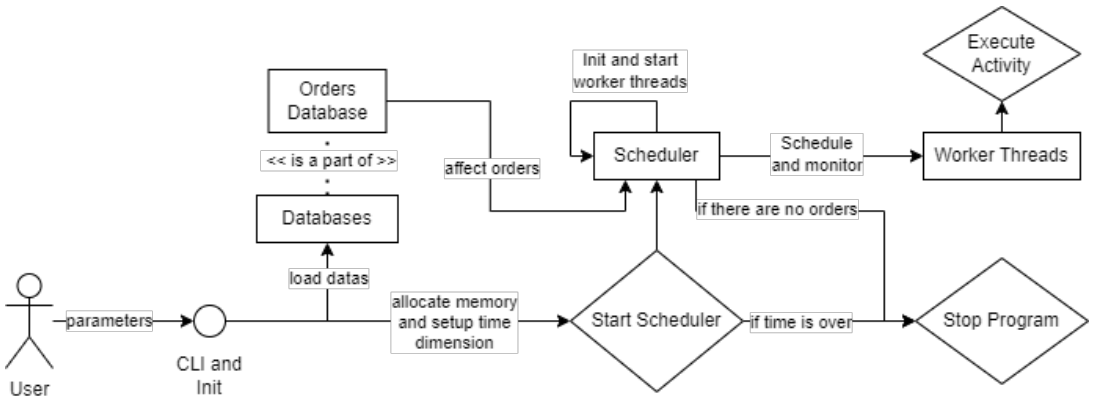

Figure 9: Diagram of the global simulation workflow.

4 Feedback and Conclusion

The development of this simulation has allowed for the implementation of many essential features to represent a real-time logistics platform. However, some parts of the logic, although identified, could not be fully implemented due to time constraints.

First, the logic related to **customer delivery** could not be finalized. This module, although conceptually integrated, requires further development to simulate the complete process, from the departure of goods to their reception by the customer. Additionally, the implementation of a **graphical display of results** remains an important improvement to offer a more intuitive and detailed visualization of the simulated processes.

Despite these limitations, the main objective of the simulation, namely the implementation of **real-time logic**, has been achieved. This includes the efficient management of task concurrency and optimized use of memory resources, ensuring performance suitable for a complex simulation.

Finally, a **solid general logic** has been implemented, integrating different dimensions (temporal and spatial) as well as modular design patterns, such as the *Strategy* pattern and state machines. These foundations provide a robust basis for future extensions and allow for the consideration of adding new features with relative ease.

Appendix: Class Diagram

The class diagram presented below illustrates the structure and relationships between the different components of the simulated system. It provides an overview of dependencies and the modularity of the architecture.

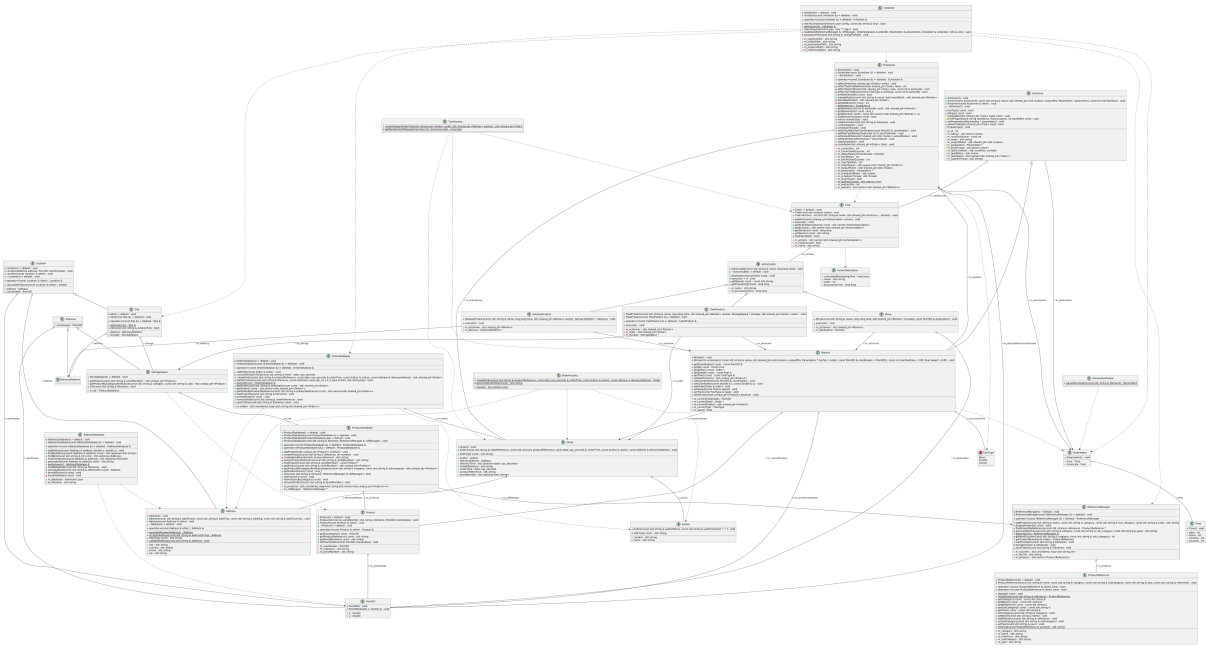

Figure 10: Class diagram of the system.